|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Java SE 8 |  |  |  |  |
| =   |  | | --- | | Java SE 8 | |  | March 18, 2014 |  | JSR 335, JEP 126: Language-level support for lambda expressions (officially, lambda expressions; unofficially, closures) under Project Lambda[243] and default methods (virtual extension methods)[244][245][246] which allow the addition of methods to interfaces without breaking existing implementations. There was an ongoing debate in the Java community on whether to add support for lambda expressions.[247][248] Sun later declared that lambda expressions would be included in Java and asked for community input to refine the feature.[249] Supporting lambda expressions also enables functional-style operations on streams of elements, such as MapReduce-inspired transformations on collections. Default methods allow an author of an API to add new methods to an interface without breaking the old code using it. Although it was not their primary intent,[244] default methods also allow multiple inheritance of behavior (but not state). JSR 223, JEP 174: Project Nashorn, a JavaScript runtime which allows developers to embed JavaScript code within applications JSR 308, JEP 104: Annotation on Java types[250] Unsigned integer arithmetic[251] JSR 337, JEP 120: Repeating annotations[252] JSR 310, JEP 150: Date and time API[253] JEP 178: Statically-linked JNI libraries[254] JEP 153: Launch JavaFX applications (direct launching of JavaFX application JARs)[255] JEP 122: Remove the permanent generation[256] |
|  |  |  |  |  |

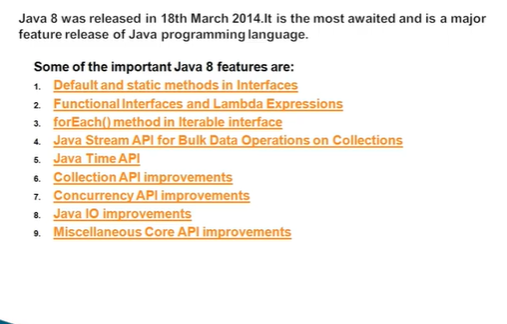
‘

‘

|  |
| --- |
| Java SE 8 |

# JAVA 8 Features details





# Interface Extension methods

## Default and static methods in Java 8 Interface\_PART1

## Why default method?

For example, if several classes such as A, B, C and D implements an interface XYZInterface then if we add a new method to the XYZInterface, we have to change the code in all the classes(A, B, C and D) that implements this interface. In this example we have only four classes that implements the interface which we want to change but imagine if there are hundreds of classes implementing an interface then it would be almost impossible to change the code in all those classes. This is why in java 8, we have a new concept “default methods”. These methods can be added to any existing interface and we do not need to implement these methods in the implementation classes mandatorily, thus we can add these default methods to existing interfaces without breaking the code.

We can say that concept of default method is introduced in java 8 to add the new methods in the existing interfaces in such a way so that they are backward compatible. Backward compatibility is adding new features without breaking the old code.

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

## ****3. Default Interface Methods in Action****

### Scenario One: Default Method in Interface

To understand using default method, I am creating an interface Village that has some method declarations and one default method. Default method starts with default keyword. *By default all methods of interface will be public, so no need to use public keyword to declare and define methods in interface.*   
**Village.java**

package com.concretepage;

public interface Village {

void setNumOfPeople(int num);

void setName(String name);

default String getBusinessType(){

return "Most of the Village people do Farming";

}

}

Create a Location class that will implement Village interface. Default method will automatically be available in this class.   
**Location.java**

package com.concretepage;

public class Location implements Village {

public int noOfPeople;

public String name;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

}

To test the scenario, create a Main class and access default method by Location object.   
**Main.java**

package com.concretepage;

public class Main {

public static void main(String[] args){

Location lo = new Location();

System.out.println(lo.getBusinessType());

}

}

Output will be as below.

Most of the Village people do Farming

### Scenario Two: Static Method in Interface

Now we can write static method in interface too. In our Village interface, I have declared getVillageId() as an static method. This static method can be accessed in default method as well.   
**Village.java**

package com.concretepage;

public interface Village {

void setNumOfPeople(int num);

void setName(String name);

static int getVillageId(){

return 1;

}

default String getBusinessType(){

return "Business type is Farming and village id:"+getVillageId();

}

}

I am doing some changes in Location class to use static method. We can use static method by interface name.   
**Location.java**

package com.concretepage;

public class Location implements Village {

public int noOfPeople;

public String name;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

public int getLocationId(){

return Village.getVillageId();

}

}

Find the main method to test the static method.   
**Main.java**

package com.concretepage;

public class Main {

public static void main(String[] args){

Location lo = new Location();

System.out.println(lo.getBusinessType());

System.out.println("Village id:"+Village.getVillageId());

System.out.println("Location Id:"+lo.getLocationId());

}

}

Run the Main class and check the output.

Business type is Farming and village id:1

Village id:1

Location Id:1

### Scenario Three: Multiple Inheritance- Default Method with Same Name in Two Interfaces

In multiple inheritance scenarios, where a class implements more than one interface, we need to check how default method behaves. Now I am creating one more interface that contains getBusinessType() as default method.   
**City.java**

package com.concretepage;

public interface City {

void setName(String name);

void setArea(int area);

default String getBusinessType(){

return "Service";

}

}

For multiple inheritances, Location class will implement Village and City interfaces both. As Village and City both contains default method with same name, so because of ambiguity, the Location class will force to define that default method explicitly in the class. The Location class will not compile until we define a method with same name as default method.   
**Location.java**

package com.concretepage;

public class Location implements Village, City {

public int noOfPeople;

public String name;

public int area;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

@Override

public void setArea(int area){

this.area = area;

}

@Override

public String getBusinessType(){

return "People do business like Farming and Service.";

}

public int getLocationId(){

return Village.getVillageId();

}

}

Run the Main class and the output will be as below.

People do business like Farming and Service.

Village id:1

Location Id:1

## Default Method and Multiple Inheritance

The [multiple inheritance](https://beginnersbook.com/2013/05/java-multiple-inheritance/) problem can occur, when we have two interfaces with the default methods of same signature. Lets take an example.

interface MyInterface{

default void newMethod(){

System.out.println("Newly added default method");

}

void existingMethod(String str);

}

interface MyInterface2{

default void newMethod(){

System.out.println("Newly added default method");

}

void disp(String str);

}

public class Example implements MyInterface, MyInterface2{

// implementing abstract methods

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public void disp(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

}

}

Output:

Error: Duplicate default methods named newMethod with the parameters () and () are inherited from the types MyInterface2 and MyInterface

This is because we have the same method in both the interface and the compiler is not sure which method to be invoked.

## How conflicts are resolved while calling default methods?

So far so good. We have got all basics well. Now move to complicated things. In java, a class can implement N number of interface. Additionally, a interface can also extend another interface as well. An if any default method is declared in two such interfaces which are implemented by single class. then obviously class will get confused which method to call.

**Rules for this conflict resolution are as follows:**

**1)** Most preferred are the overridden methods in classes. They will be matched and called if found before matching anything.  
**2)** The method with the same signature in the “most specific default-providing interface” is selected. This means if class Animal implements two interfaces i.e. Moveable and Walkable such that Walkable extends Moveable. Then Walkable is here most specific interface and default method will be chosen from here if method signature is matched.  
**3)** If Moveable and Walkable are independent interfaces then a serious conflict condition happen, and compiler will complain then it is unable to decide. The you have to help compiler by providing extra info that from which interface the default method should be called. e.g.

|  |
| --- |
| Walkable.super.move();  //or  Moveable.super.move(); |

PROCEDURE 2

For multiple inheritances, Location class will implement Village and City interfaces both. As Village and City both contains default method with same name, so because of ambiguity, the Location class will force to define that default method explicitly in the class. The Location class will not compile until we define a method with same name as default method.   
**Location.java**

package com.concretepage;

public class Location implements Village, City {

public int noOfPeople;

public String name;

public int area;

@Override

public void setNumOfPeople(int n){

this.noOfPeople = n;

}

@Override

public void setName(String name){

this.name = name;

}

@Override

public void setArea(int area){

this.area = area;

}

@Override

public String getBusinessType(){

return "People do business like Farming and Service.";

}

public int getLocationId(){

return Village.getVillageId();

}

}

Run the Main class and the output will be as below.

People do business like Farming and Service.

Village id:1

Location Id:1

## ****5. Static Interface Methods****

Aside from being able to declare default methods in interfaces, **Java 8 allows us to define and implement staticmethods in interfaces**.

Since static methods don’t belong to a particular object, they are not part of the API of the classes implementing the interface, and they have to be **called by using the interface name preceding the method name**.

To understand how static methods work in interfaces, let’s refactor the Vehicle interface and add to it a static utility method:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public interface Vehicle {        // regular / default interface methods        static int getHorsePower(int rpm, int torque) {          return (rpm \* torque) / 5252;      }  } |

**Defining a static method within an interface is identical to defining one in a class.** Moreover, a static method can be invoked within other static and default methods.

Now, say that we want to calculate the [horsepower](https://en.wikipedia.org/wiki/Horsepower) of a given vehicle’s engine. We just call the getHorsePower()method:

|  |  |
| --- | --- |
| 1 | Vehicle.getHorsePower(2500, 480)); |

The idea behind static interface methods is to provide a simple mechanism that allows us to **increase the degree of**[**cohesion**](https://en.wikipedia.org/wiki/Cohesion_(computer_science)) of a design by putting together related methods in one single place without having to create an object.

Pretty much **the same can be done with abstract classes.** The main difference lies in the fact that **abstract classes can have constructors, state, and behavior**.

Furthermore, static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes that are simply placeholders for static methods.

## Java 8 Example: Static method in Interface

As mentioned above, the static methods in interface are similar to default method so we need not to implement them in the implementation classes. We can safely add them to the existing interfaces without changing the code in the implementation classes. Since these methods are static, we cannot override them in the implementation classes.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Output:

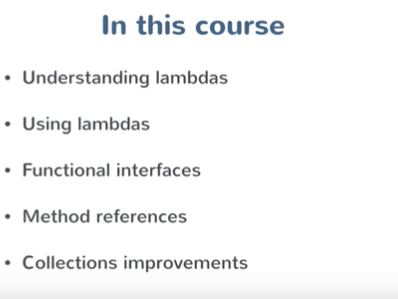
Newly added default method

Newly added static method

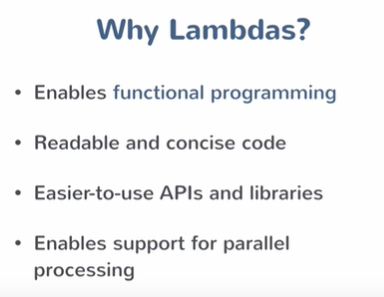
String is: Java 8 is easy to learn

# Lambda Expressions

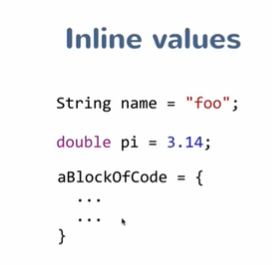
## Java 8 Lambda Basics 1 - Introduction



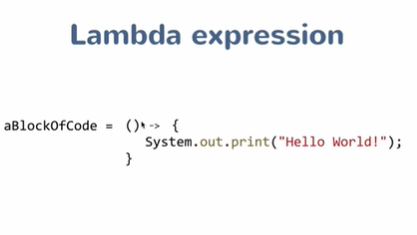
### Why lambdas?

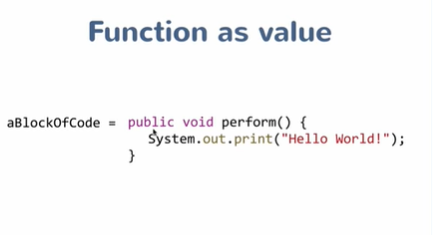


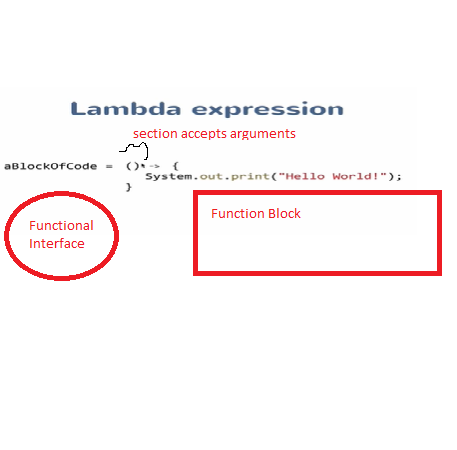
What is Inline values



### Lambda Expression







# Java Lambda Expressions

* [Java Lambdas and the Single Method Interface](http://tutorials.jenkov.com/java/lambda-expressions.html#single-method-interface)
  + [Matching Lambdas to Interfaces](http://tutorials.jenkov.com/java/lambda-expressions.html#matching-lambdas-to-interfaces)
  + [Interfaces With Default and Static Methods](http://tutorials.jenkov.com/java/lambda-expressions.html#interfaces-with-default-and-static-methods)
* [Lambda Expressions vs. Anonymous Interface Implementations](http://tutorials.jenkov.com/java/lambda-expressions.html#lambda-expressions-vs-anonymous-interface-implementations)
* [Lambda Type Inference](http://tutorials.jenkov.com/java/lambda-expressions.html#type-inference)
* [Lambda Parameters](http://tutorials.jenkov.com/java/lambda-expressions.html#lambda-parameters)
  + [Zero Parameters](http://tutorials.jenkov.com/java/lambda-expressions.html#zero-parameter)
  + [One Parameter](http://tutorials.jenkov.com/java/lambda-expressions.html#one-parameter)
  + [Multiple Parameters](http://tutorials.jenkov.com/java/lambda-expressions.html#multiple-parameters)
  + [Parameter Types](http://tutorials.jenkov.com/java/lambda-expressions.html#parameter-types)
  + [var Parameter Types from Java 11](http://tutorials.jenkov.com/java/lambda-expressions.html#var-parameter-types-java-11)
* [Lambda Function Body](http://tutorials.jenkov.com/java/lambda-expressions.html#lambda-body)
* [Returning a Value From a Lambda Expression](http://tutorials.jenkov.com/java/lambda-expressions.html#returning%20values-from-lambda-expression)
* [Lambdas as Objects](http://tutorials.jenkov.com/java/lambda-expressions.html#lambdas-as-objects)
* [Variable Capture](http://tutorials.jenkov.com/java/lambda-expressions.html#variable-capture)
  + [Local Variable Capture](http://tutorials.jenkov.com/java/lambda-expressions.html#local-variable-capture)
  + [Instance Variable Capture](http://tutorials.jenkov.com/java/lambda-expressions.html#instance-variable-capture)
  + [Static Variable Capture](http://tutorials.jenkov.com/java/lambda-expressions.html#static-variable-capture)
* [Method References as Lambdas](http://tutorials.jenkov.com/java/lambda-expressions.html#method-references-as-lambdas)
  + [Static Method References](http://tutorials.jenkov.com/java/lambda-expressions.html#static-method-references)
  + [Parameter Method Reference](http://tutorials.jenkov.com/java/lambda-expressions.html#parameter-method-reference)
  + [Instance Method References](http://tutorials.jenkov.com/java/lambda-expressions.html#instance-method-references)
  + [Constructor References](http://tutorials.jenkov.com/java/lambda-expressions.html#constructor-references)

|  |  |
| --- | --- |
|  | Jakob Jenkov Last update: 2019-01-18 |

Java lambda expressions are new in Java 8. Java lambda expressions are Java's first step into functional programming. A Java lambda expression is thus a function which can be created without belonging to any class. A Java lambda expression can be passed around as if it was an object and executed on demand.

Java lambda expressions are commonly used to implement simple event listeners / callbacks, or in functional programming with the [**Java Streams API**](http://tutorials.jenkov.com/java-collections/streams.html).

If you prefer video, I have a video version of this tutorial in this [**Java Lambda Expression YouTube Playlist**](https://www.youtube.com/watch?v=lIXs4Y8sJCk&list=PLL8woMHwr36HQhhPPdV_T8rigbuywMpD7). Here is the first video in this playlist:

## Java Lambdas and the Single Method Interface

Functional programming is very often used to implement event listeners. Event listeners in Java are often defined as Java interfaces with a single method. Here is a fictive single method interface example:

public interface StateChangeListener {

public void onStateChange(State oldState, State newState);

}

This Java interface defines a single method which is called whenever the state changes (in whatever is being observed).

In Java 7 you would have to implement this interface in order to listen for state changes. Imagine you have a class called StateOwner which can register state event listeners. Here is an example:

public class StateOwner {

public void addStateListener(StateChangeListener listener) { ... }

}

In Java 7 you could add an event listener using an anonymous interface implementation, like this:

StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(new StateChangeListener() {

public void onStateChange(State oldState, State newState) {

// do something with the old and new state.

}

});

First a StateOwner instance is created. Then an anonymous implementation of the StateChangeListener interface is added as listener on the StateOwner instance.

In Java 8 you can add an event listener using a Java lambda expression, like this:

StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(

**(oldState, newState) -> System.out.println("State changed")**

);

The lambda expressions is this part:

(oldState, newState) -> System.out.println("State changed")

The lambda expression is matched against the parameter type of the addStateListener() method's parameter. If the lambda expression matches the parameter type (in this case the StateChangeListener interface) , then the lambda expression is turned into a function that implements the same interface as that parameter.

Java lambda expressions can only be used where the type they are matched against is a single method interface. In the example above, a lambda expression is used as parameter where the parameter type was the StateChangeListener interface. This interface only has a single method. Thus, the lambda expression is matched successfully against that interface.

### Matching Lambdas to Interfaces

A single method interface is also sometimes referred to as a functional interface. Matching a Java lambda expression against a functional interface is divided into these steps:

* Does the interface have only one abstract (unimplemented) method?
* Does the parameters of the lambda expression match the parameters of the single method?
* Does the return type of the lambda expression match the return type of the single method?

If the answer is yes to these three questions, then the given lambda expression is matched successfully against the interface.

### Interfaces With Default and Static Methods

From Java 8 a [**Java interface**](http://tutorials.jenkov.com/java/interfaces.html) can contain both default methods and static methods. Both default methods and static methods have an implementation defined directly in the interface declaration. This means, that a Java lambda expression can implement interfaces with more than one method - as long as the interface only has a single unimplemented (AKA abstract) method.

In other words, an interface is still a functional interface even if it contains default and static methods, as long as the interface only contains a single unimplemented (abstract) method. Here is a video version of this little section:

The following interface can be implemented with a lambda expression:

import java.io.IOException;

import java.io.OutputStream;

public interface MyInterface {

void printIt(String text);

default public void printUtf8To(String text, OutputStream outputStream){

try {

outputStream.write(text.getBytes("UTF-8"));

} catch (IOException e) {

throw new RuntimeException("Error writing String as UTF-8 to OutputStream", e);

}

}

static void printItToSystemOut(String text){

System.out.println(text);

}

}

Even though this interface contains 3 methods it can be implemented by a lambda expression, because only one of the methods is unimplemented. Here is how the implementation looks:

MyInterface myInterface = (String text) -> {

System.out.print(text);

};

## Lambda Expressions vs. Anonymous Interface Implementations

Even though lambda expressions are close to anonymous interface implementations, there are a few differences that are worth noting.

The major difference is, that an anonymous interface implementation can have state (member variables) whereas a lambda expression cannot. Look at this interface:

public interface MyEventConsumer {

public void consume(Object event);

}

This interface can be implemented using an anonymous interface implementation, like this:

MyEventConsumer consumer = new MyEventConsumer() {

public void consume(Object event){

System.out.println(event.toString() + " consumed");

}

};

This anonymous MyEventConsumer implementation can have its own internal state. Look at this redesign:

MyEventConsumer myEventConsumer = new MyEventConsumer() {

private int eventCount = 0;

public void consume(Object event) {

System.out.println(event.toString() + " consumed " + this.eventCount++ + " times.");

}

};

Notice how the anonymous MyEventConsumer implementation now has a field named eventCount.

A lambda expression cannot have such fields. A lambda expression is thus said to be stateless.

## Lambda Type Inference

Before Java 8 you would have to specify what interface to implement, when making anonymous interface implementations. Here is the anonymous interface implementation example from the beginning of this text:

stateOwner.addStateListener(new StateChangeListener() {

public void onStateChange(State oldState, State newState) {

// do something with the old and new state.

}

});

With lambda expressions the type can often be *inferred* from the surrounding code. For instance, the interface type of the parameter can be inferred from the method declaration of the addStateListener() method (the single method on the StateChangeListener interface). This is called *type inference*. The compiler infers the type of a parameter by looking elsewhere for the type - in this case the method definition. Here is the example from the beginning of this text, showing that the StateChangeListener interface is not mentioned in the lambda expression:

stateOwner.addStateListener(

(oldState, newState) -> System.out.println("State changed")

);

In the lambda expression the parameter types can often be inferred too. In the example above, the compiler can infer their type from the onStateChange() method declaration. Thus, the type of the parameters oldState and newState are inferred from the method declaration of the onStateChange() method.

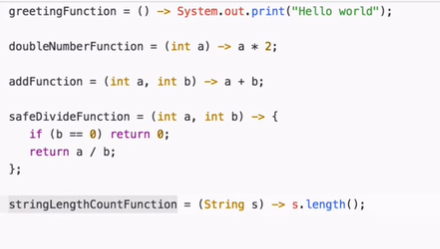
## Lambda Parameters

Since Java lambda expressions are effectively just methods, lambda expressions can take parameters just like methods. The (oldState, newState) part of the lambda expression shown earlier specifies the parameters the lambda expression takes. These parameters have to match the parameters of the method on the single method interface. In this case, these parameters have to match the parameters of the onStateChange() method of the StateChangeListener interface:

public void onStateChange(State oldState, State newState);

As a minimum the number of parameters in the lambda expression and the method must match.

Second, if you have specified any parameter types in the lambda expression, these types must match too. I haven't shown you how to put types on lambda expression parameters yet (it is shown later in this text), but in many cases you don't need them.



### Zero Parameters

If the method you are matching your lambda expression against takes no parameters, then you can write your lambda expression like this:

() -> System.out.println("Zero parameter lambda");

Notice how the parentheses have no content in between. That is to signal that the lambda takes no parameters.

### One Parameter

If the method you are matching your Java lambda expression against takes one parameter, you can write the lambda expression like this:

(param) -> System.out.println("One parameter: " + param);

Notice the parameter is listed inside the parentheses.

When a lambda expression takes a single parameter, you can also omit the parentheses, like this:

param -> System.out.println("One parameter: " + param);

### Multiple Parameters

If the method you match your Java lambda expression against takes multiple parameters, the parameters need to be listed inside parentheses. Here is how that looks in Java code:

(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);

Only when the method takes a single parameter can the parentheses be omitted.

### Parameter Types

Specifying parameter types for a lambda expression may sometimes be necessary if the compiler cannot infer the parameter types from the functional interface method the lambda is matching. Don't worry, the compiler will tell you when that is the case. Here is a Java lambda parameter type example:

(Car car) -> System.out.println("The car is: " + car.getName());

As you can see, the type (Car) of the car parameter is written in front of the parameter name itself, just like you would when declaring a parameter in a method elsewhere, or when making an anonymous implementation of an interface.

### var Parameter Types from Java 11

From Java 11 you can use the var keyword as parameter type. The var keyword was introduced in Java 10 as [**local variable type inference**](http://tutorials.jenkov.com/java/variables.html#java-local-variable-type-inference). From Java 11 var can also be used for lambda parameter types. Here is an example of using the Java var keyword as parameter types in a lambda expression:

Function<String, String> toLowerCase = (var input) -> input.toLowerCase();

The type of the parameter declared with the var keyword above will be inferred to the type String, because the type declaration of the variable has its generic type set to Function<String, String>, which means that the parameter type and return type of the Function is String.

## Lambda Function Body

The body of a lambda expression, and thus the body of the function / method it represents, is specified to the right of the -> in the lambda declaration: Here is an example:

(oldState, newState) -> **System.out.println("State changed")**

If your lambda expression needs to consist of multiple lines, you can enclose the lambda function body inside the { } bracket which Java also requires when declaring methods elsewhere. Here is an example:

(oldState, newState) -> {

System.out.println("Old state: " + oldState);

System.out.println("New state: " + newState);

}

## Returning a Value From a Lambda Expression

You can return values from Java lambda expressions, just like you can from a method. You just add a return statement to the lambda function body, like this:

(param) -> {

System.out.println("param: " + param);

return "return value";

}

In case all your lambda expression is doing is to calculate a return value and return it, you can specify the return value in a shorter way. Instead of this:

(a1, a2) -> { return a1 > a2; }

You can write:

(a1, a2) -> a1 > a2;

The compiler then figures out that the expression a1 > a2 is the return value of the lambda expression (hence the name lambda *expressions* - as expressions return a value of some kind).

## Lambdas as Objects

A Java lambda expression is essentially an object. You can assign a lambda expression to a variable and pass it around, like you do with any other object. Here is an example:

public interface MyComparator {

public boolean compare(int a1, int a2);

}

MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);

The first code block shows the interface which the lambda expression implements. The second code block shows the definition of the lambda expression, how the lambda expression is assigned to variable, and finally how the lambda expression is invoked by invoking the interface method it implements.

## Variable Capture

A Java lambda expression is capable of accessing variables declared outside the lambda function body under certain circumstances. I have a video version of this section here:

Java lambdas can capture the following types of variables:

* Local variables
* Instance variables
* Static variables

Each of these variable captures will described in the following sections.

### Local Variable Capture

A Java lambda can capture the value of a local variable declared outside the lambda body. To illustrate that, first look at this single method interface:

public interface MyFactory {

public String create(char[] chars);

}

Now, look this lambda expression that implements the MyFactory interface:

MyFactory myFactory = (chars) -> {

return new String(chars);

};

Right now this lambda expression is only referencing the parameter value passed to it (chars). But we can change that. Here is an updated version that references a String variable declared outside the lambda function body:

String myString = "Test";

MyFactory myFactory = (chars) -> {

return myString + ":" + new String(chars);

};

As you can see, the lambda body now references the local variable myString which is declared outside the lambda body. This is possible if, and only if, the variable being references is "effectively final", meaning it does not change its value after being assigned. If the myString variable had its value changed later, the compiler would complain about the reference to it from inside the lambda body.

### Instance Variable Capture

A lambda expression can also capture an instance variable in the object that creates the lambda. Here is an example that shows that:

public class EventConsumerImpl {

private String name = "MyConsumer";

public void attach(MyEventProducer eventProducer){

eventProducer.listen(e -> {

System.out.println(this.name);

});

}

}

Notice the reference to this.name inside the lambda body. This captures the name instance variable of the enclosing EventConsumerImpl object. It is even possible to change the value of the instance variable after its capture - and the value will be reflected inside the lambda.

The semantics of this is actually one of the areas where Java lambdas differ from anonymous implementations of interfaces. An anonymous interface implementation can have its own instance variables which are referenced via the this reference. However, an lambda cannot have its own instance variables, so this always points to the enclosing object.

Note: The above design of an event consumer is not particularly elegant. I just made it like that to be able to illustrate instance variable capture.

### Static Variable Capture

A Java lambda expression can also capture static variables. This is not surprising, as static variables are reachable from everywhere in a Java application, provided the static variable is accessible (packaged scoped or public).

Here is an example class that creates a lambda which references a static variable from inside the lambda body:

public class EventConsumerImpl {

private static String someStaticVar = "Some text";

public void attach(MyEventProducer eventProducer){

eventProducer.listen(e -> {

System.out.println(someStaticVar);

});

}

}

The value of a static variable is also allowed to change after the lambda has captured it.

Again, the above class design is a bit nonsensical. Don't think too much about that. The class primarily serves to show you that a lambda can access static variables.

## Method References as Lambdas

In the case where all your lambda expression does is to call another method with the parameters passed to the lambda, the Java lambda implementation provides a shorter way to express the method call. First, here is an example single function interface:

public interface MyPrinter{

public void print(String s);

}

And here is an example of creating a Java lambda instance implementing the MyPrinter interface:

MyPrinter myPrinter = (s) -> { System.out.println(s); };

Because the lambda body only consists of a single statement, we can actually omit the enclosing { } brackets. Also, since there is only one parameter for the lambda method, we can omit the enclosing ( ) brackets around the parameter. Here is how the resulting lambda declaration looks:

MyPrinter myPrinter = s -> System.out.println(s);

Since all the lambda body does is forward the string parameter to the System.out.println() method, we can replace the above lambda declaration with a method reference. Here is how a lambda method reference looks:

MyPrinter myPrinter = System.out::println;

Notice the double colons :: . These signal to the Java compiler that this is a method reference. The method referenced is what comes after the double colons. Whatever class or object that owns the referenced method comes before the double colons.

You can reference the following types of methods:

* Static method
* Instance method on parameter objects
* Instance method
* Constructor

Each of these types of method references are covered in the following sections.

### Static Method References

The easiest methods to reference are static methods. Here is first an example of a single function interface:

public interface Finder {

public int find(String s1, String s2);

}

And here is a static method that we want to create a method reference to:

public class MyClass{

public static int doFind(String s1, String s2){

return s1.lastIndexOf(s2);

}

}

And finally here is a Java lambda expression referencing the static method:

Finder finder = MyClass::doFind;

Since the parameters of the Finder.find() and MyClass.doFind() methods match, it is possible to create a lambda expression that implements Finder.find() and references the MyClass.doFind() method.

### Parameter Method Reference

You can also reference a method of one of the parameters to the lambda. Imagine a single function interface that looks like this:

public interface Finder {

public int find(String s1, String s2);

}

The interface is intended to represent a component able to search s1 for occurrences of s2. Here is an example of a Java lambda expression that calls String.indexOf() to search:

Finder finder = String::indexOf;

This is equivalent of this lambda definition:

Finder finder = (s1, s2) -> s1.indexOf(s2);

Notice how the shortcut version references a single method. The Java compiler will attempt to match the referenced method against the first parameter type, using the second parameter type as parameter to the referenced method.

### Instance Method References

Third, it is also possible to reference an instance method from a lambda definition. First, let us look at a single method interface definition:

public interface Deserializer {

public int deserialize(String v1);

}

This interface represents a component that is capable of "deserializing" a String into an int.

Now look at this StringConverter class:

public class StringConverter {

public int convertToInt(String v1){

return Integer.valueOf(v1);

}

}

The convertToInt() method has the same signature as the deserialize() method of the Deserializer deserialize() method. Because of that, we can create an instance of StringConverter and reference its convertToInt() method from a Java lambda expression, like this:

StringConverter stringConverter = new StringConverter();

Deserializer des = stringConverter::convertToInt;

The lambda expression created by the second of the two lines references the convertToInt method of the StringConverter instance created on the first line.

### Constructor References

Finally it is possible to reference a constructor of a class. You do that by writing the class name followed by ::new, like this:

MyClass::new

Too see how to use a constructor as a lambda expression, look at this interface definition:

public interface Factory {

public String create(char[] val);

}

The create() method of this interface matches the signature of one of the constructors in the String class. Therefore this constructor can be used as a lambda. Here is an example of how that looks:

Factory factory = String::new;

This is equivalent to this Java lambda expression:

Factory factory = chars -> new String(chars);

## ****Functional Interface****:

The functional interface is an interface with one abstract method, but it can have any number of default and static methods. The runnable and comparable interface are examples of a functional interface.

 (These were previously called SAM Types, which stood for "Single Abstract Method".)

Example:

public interface TestInterface {

public abstract void show(int n);

public static void hello() {

System.out.println("static method");

}

public default void test() {

System.out.println("default method");

}

}

### @FunctionalInterface Annotation:

We use the **@Functional annotation** with a functional interface. This ensures that the interface can’t have more than one abstract method. If we define more than one abstract method in this interface, then the compiler shows an error ‘*Invalid '@FunctionalInterface' annotation.*’

@FunctionalInterface

public interface TestInterface {

public abstract void show(int n);

public static void hello() {

System.out.println("static method");

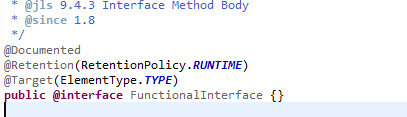
}

public default void test() {

System.out.println("default method");

}

}



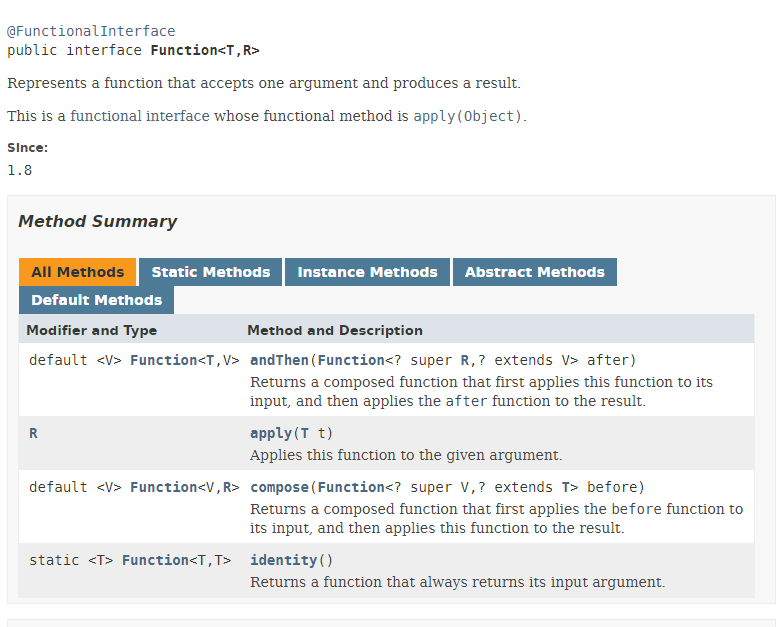
Examples of functional interface in java8

* [**java.lang.Runnable**](http://download.oracle.com/javase/7/docs/api/java/lang/Runnable.html)
* [**java.util.concurrent.Callable**](http://download.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html)
* [**java.security.PrivilegedAction**](http://download.oracle.com/javase/7/docs/api/java/security/PrivilegedAction.html)
* [**java.util.Comparator**](http://download.oracle.com/javase/7/docs/api/java/util/Comparator.html)
* [**java.io.FileFilter**](http://download.oracle.com/javase/7/docs/api/java/io/FileFilter.html)
* [**java.beans.PropertyChangeListener**](http://www.fxfrog.com/docs_www/api/java/beans/PropertyChangeListener.html)

In addition, Java SE 8 adds a new package, [**java.util.function**](http://download.java.net/jdk8/docs/api/java/util/function/package-summary.html), which contains functional interfaces that are expected to be commonly used, such as:

* **Predicate<T>** -- a boolean-valued property of an object
* **Consumer<T>** -- an action to be performed on an object
* **Function<T,R>** -- a function transforming a T to a R
* **Supplier<T>** -- provide an instance of a T (such as a factory)
* **UnaryOperator<T>** -- a function from T to T
* **BinaryOperator<T>** -- a function from (T, T) to T





### Function

What is java.util.function.Function

**Function<T, R>** is an in-built functional interface introduced in Java 8 in the **java.util.function** package. The primary purpose for which **Function<T, R>** has been created is for mapping scenarios i.e when an object of a type is taken as input and it is converted(or mapped) to another type. Common usage of Function is in streams where-in the map function of a stream accepts an instance of Function to convert the stream of one type to a stream of another type.  
Since **Function<T, R>** is a [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/), hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Function Descriptor of Function<T, R>**  
**Function<T, R>**’s Function Descriptor is **T -> R**. This means an object of type T is input to the lambda and an object of type R is obtained as return value. To understand Function Descriptors in details you can refer the [function descriptor tutorial](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/).

**Advantage of predefined java.util.function.Function**: In all scenarios where an object of a particular type is the input, an operation is performed on it and and object of another type is returned as output, the in-built functional interface **Function<T, R>** can be used without the need to define a new functional interface every time.

**java.util.function.Function source code**

**java.util.function.Function source code**

|  |
| --- |
| @FunctionalInterface  public interface Function<T, R> {      R apply(T t);      default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {          Objects.requireNonNull(before);          return (V v) -> apply(before.apply(v));      }      default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {          Objects.requireNonNull(after);          return (T t) -> after.apply(apply(t));      }      static <T> Function<T, T> identity() {          return t -> t;      }  } |

**Salient Points regarding Function<T, R>’s source code**

* **Function** interface has been defined with the generic types **T** & **R**, where **T** is the type of the input and **R** is the output type.
* Method**apply()** is the primary abstract functional method of **Function** interface. It takes as input a parameter **t** of type **T** and gives an output object of type **R**.
* **Function<T, R>** has two default methods. First default method **compose()** combines the function on which it is applied(lets call it the current function) with another function, named **before**, in such a way that when the combined function is applied then first the **before** function is applied which converts the input type **V** to type **T**. And then the current function converts this object of type **T** to its output type **R**. Thus, the combined function obtained as a result of **compose()** applies both the functions, in the process converting type **V** to **R**.
* The second default method is **andThen()** which combines the function on which it is applied(current function) with another function, named **after**, in such a way that when the combined function is called then first the current function is applied which converts the input type **T** to type **R**. And then the **after** function is applied which converts from type **R** to **V**. Thus, the combined function obtained by using **andThen()** default method applies both functions internally, in the process converting type **T** to type **V**.
* **Function<T, R>** also contains a static method **identity()** which is very simple as it returns as-is whatever is given to it as input. In the code above it takes as input a parameter **t** of Type **T** and returns back this **t**.

**Usage of apply() method of Function**  
The below code shows example usage of **apply()** method where it converts/maps from a list of Employee types to a list of Strings containing the names of all Employees. Let us now go through the code after which I will explain how the **apply()** method works-

**Java 8 code example showing usage of Function.apply() method**

|  |
| --- |
| package com.javabrahman.java8;  import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import java.util.function.Function;  public class FunctionTRExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      List<Employee> employeeList=       Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));      List<String> empNameList=convertEmpListToNamesList(employeeList, funcEmpToString);      empNameList.forEach(System.out::println);   }   public static List<String> convertEmpListToNamesList(List<Employee> employeeList, Function<Employee, String> funcEmpToString){     List<String> empNameList=new ArrayList<String>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToString.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

Tom Jones  
Harry Major  
Ethan Hardy  
Nancy Smith  
Deborah Sprightly

**Explanation of above example’s Code & Output**

* **funcEmpToString**is an instance of **Function<Employee,String>**. This is the **java.util.function.Function** instance which is used to convert/map from an **Employee**object to a **String**value.
* The lambda defining funcEmpToString is – **(Employee e)-> {return e.getName();}** . It takes as input an **Employee** object and returns his\her name, which is a **String**value, as output.
* The list of employees is passed to method **convertEmpListToNamesList()** along with the Function object **funcEmpToString**;
* The method **convertEmpListToNamesList()** iterates over all the employees in the employee list, applies the function **funcEmpToString** to each of the **Employee** objects, getting back the employee names in **String** format, which it puts in a employee name list and sends it back to the **main()** method.
* On printing the employee name list we get the names of all the employees as required.

**Usage of default method andThen() of Function**  
As explained earlier, **andThen()** default method combines the current Function instance with another one and returns a combined Function instance which applies the two functions in sequence with the function passed as parameter to **andThen()** being invoked after the current function.  
Lets see the example below which uses the same **funcEmpToString** Function as used in the **apply()** method’s usage example above and combines it with a **initialFunction** Function instance which maps\converts a String to its first letter –

**Java 8 code showing usage of default method Function.andThen()**

|  |
| --- |
| //import statements are same as in apply() example  public class FunctionTRAndThenExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      List<Employee> employeeList=       Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));      Function<String,String> initialFunction= (String s)->s.substring(0,1);      List<String> empNameListInitials=convertEmpListToNamesList(employeeList, funcEmpToString.andThen(initialFunction));      empNameListInitials.forEach(str->{System.out.print(" "+str);});   }    public static List<String> convertEmpListToNamesList(List<Employee> employeeList, Function<Employee, String> funcEmpToString){     List<String> empNameList=new ArrayList<String>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToString.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

T H E N D

**Explanation of above example’s Code & Output**

* Function instance **funcEmpToString** maps\converts an **Employee** object to a **String** of his\her name.
* Function instance **initialFunction** maps\converts a **String** to its initial or first letter.
* Default method **andThen()** is used to combine **initialFunction** with **funcEmpToString**. What the combined method does is that it first maps an **Employee** to his\her name and then takes out the first letter from the name as a **String** value. This combined function is passed as **Function** parameter to **convertEmpListToNamesList()** method along with the employee list.
* When the **convertEmpListToNamesList()** applies the combined function to each of the **Employee**objects, then the result is a **String** list first letters of names of each employee.
* This is the required output i.e. T H E N D

**Usage of default method compose() of Function**  
As explained earlier, **compose()** default method combines the current Function instance with another one and returns a combined Function instance which applies the two functions in sequence with the parameter function to **compose()** being invoked before the current function.  
Lets see the example below which uses the same **funcEmpToString** Function as used in the **apply()** usage example and combines it with a **funcEmpFirstName** Function instance which converts the full-name of the employee object passed to it to just the first name of the employee-

**Java 8 code showing usage of default method Function.compose()**

|  |
| --- |
| //import statements are same as in apply() example  public class FunctionTRComposeExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      Function<Employee, Employee> funcEmpFirstName=       (Employee e)-> {int index= e.getName().indexOf(" ");                   String firstName=e.getName().substring(0,index);                   e.setName(firstName);                   return e;};      List<Employee> employeeList=        Arrays.asList(new Employee("Tom Jones", 45),         new Employee("Harry Major", 25),         new Employee("Ethan Hardy", 65),         new Employee("Nancy Smith", 15),         new Employee("Deborah Sprightly", 29));      List<String> empFirstNameList= convertEmpListToNamesList(employeeList,funcEmpToString.compose(funcEmpFirstName));      empFirstNameList.forEach(str->{System.out.print(" "+str);});    }   public static List<String> convertEmpListToNamesList(List<Employee> employeeList, Function<Employee, String> funcEmpToString){     List<String> empNameList=new ArrayList<String>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToString.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

Tom Harry Ethan Nancy Deborah

**Explanation of above example’s Code & Output**

* Function instance **funcEmpToString** maps\converts an **Employee** object to a **String** value of his\her name.
* Function instance **funcEmpFirstName** maps\converts the **name** inside an **Employee** object to the first name using the **substring** method of **String**.
* Default method **compose()** is used to combine **funcEmpFirstName** with funcEmpToStringString. What the combined method does is that it first converts the name of an **Employee** into just his\her first name returning the same **Employee** object back with the changed value of **name**. It then converts\maps the **Employee**object to just its **name** as a **String**.This combined function is passed as **Function<Employee, String>**parameter to **convertEmpListToNamesList()** method along with the employee list.
* When the **convertEmpListToNamesList()** applies the combined function to each of the **Employee**objects, then the result is the list of first names of each employee.
* This is the required output i.e. Tom Harry Ethan Nancy Deborah

**Usage of static method identity() of Function**  
Static method **identity()** is very simple – it just returns back the parameter which it gets as input. Lets see an example to see how **identity()** method works –

**Java 8 code showing usage of static method Function.identity()**

|  |
| --- |
| //import statements are same as in apply() example  public class FunctionTRIdentityExample{    public static void main(String args[]){      Function<Employee, String> funcEmpToString= (Employee e)-> {return e.getName();};      List<Employee> employeeList=       Arrays.asList(new Employee("Tom Jones", 45),        new Employee("Harry Major", 25),        new Employee("Ethan Hardy", 65),        new Employee("Nancy Smith", 15),        new Employee("Deborah Sprightly", 29));      List<Employee> empNameListInitials=applyIdentityToEmpList(employeeList, Function.identity());      empNameListInitials.forEach(System.out::println);   }    public static List<Employee> applyIdentityToEmpList(List<Employee> employeeList, Function<Employee, Employee> funcEmpToEmp){     List<Employee> empNameList=new ArrayList<Employee>();     for(Employee emp:employeeList){       empNameList.add(funcEmpToEmp.apply(emp));     }     return empNameList;    }  } |

**OUTPUT of the above code**

Employee Name:Tom Jones Age:45  
Employee Name:Harry Major Age:25  
Employee Name:Ethan Hardy Age:65  
Employee Name:Nancy Smith Age:15  
Employee Name:Deborah Sprightly Age:29

**Explanation of above example’s Code & Output**

* Function instance **funcEmpToString** maps\converts an **Employee** object to a **String** of his\her name.
* Employee list is passed to the method **applyIdentityToEmpList()** along with an instance of **Function.identity()**. The parameter of **applyIdentityToEmpList()** which takes **Function.identity()** value is nothing but **Function<Employee, Employee>** i.e. an equivalent of a **Function** which takes **Employee** as input and gives back (the same) **Employee** as output.
* Method **applyIdentityToEmpList()** takes the input employee list, iterates through it, applies the **identity()** function to each employee in the list and returns back the list of employees obtained as a result of applying the **identity()** function.
* As we now know, the identity function does nothing, it just returns back the object it receives as input. So, what we get back is the same employee list which we passed to the **applyIdentityToEmpList()** method! And the same i.e. original employee list is printed as output!!
* This is how the static method **Function.identity()** works.

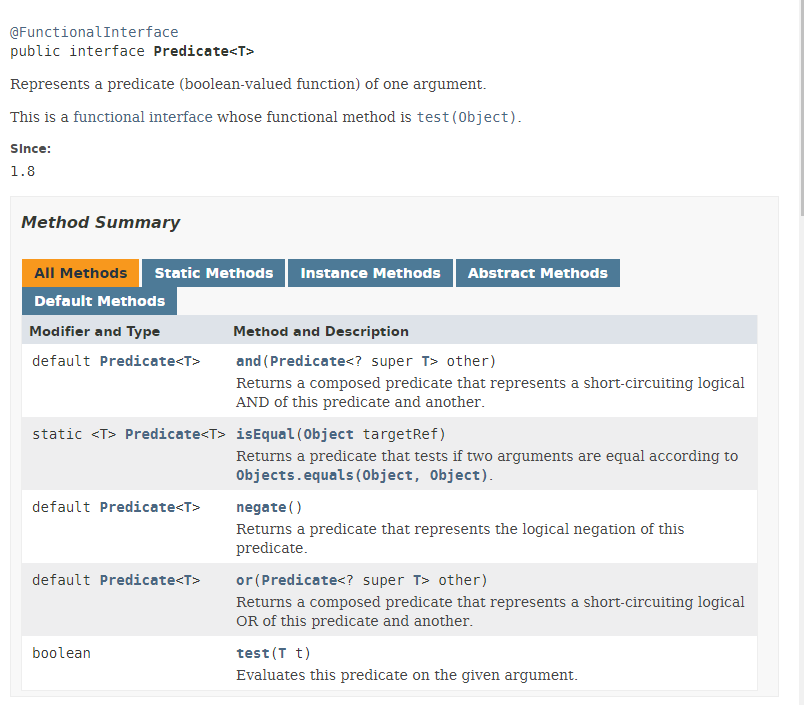
**Summary**  
In this tutorial we looked at what is the **Function<T, R>** in-built interface defined in Java 8 and what are its advantages. We then looked at how to use the Function interface using its **apply()** method, default methods **andThen()** & **compose()**, and lastly how to use the static method **identity()**.

### Predicate

Java 8 java.util.function.Predicate tutorial with examples

Predicate is a new [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) defined in java.util.function package which can be used in all the contexts where an object needs to be evaluated for a given test condition and a boolean value needs to be returned based on whether the condition was successfully met or not.  
Since Predicate is a functional interface, hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Advantage of predefined java.util.function.Predicate** – Wherever an object needs to be evaluated and a boolean value needs to be returned( or a boolean-valued Predicate exists – in mathematical terms) the Predicate functional interface can be used. The user need not define his/her own predicate-type functional interface.



Predicate is a new [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) defined in java.util.function package which can be used in all the contexts where an object needs to be evaluated for a given test condition and a boolean value needs to be returned based on whether the condition was successfully met or not.  
Since Predicate is a functional interface, hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/)or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Advantage of predefined java.util.function.Predicate** – Wherever an object needs to be evaluated and a boolean value needs to be returned( or a boolean-valued Predicate exists – in mathematical terms) the Predicate functional interface can be used. The user need not define his/her own predicate-type functional interface.

**Example of using Predicate for a boolean condition check** –

**Code showing Predicate being used for a boolean condition check**

|  |
| --- |
| import java.util.ArrayList;  import java.util.Arrays;  import java.util.List;  import java.util.function.Predicate;  public class PredicateFunctionExample{   public static void main(String args[]){    Predicate<Integer> positive = i -> i > 0;    List<Integer> integerList = Arrays.asList(                        new Integer(1),new Integer(10),                        new Integer(200),new Integer(101),                        new Integer(-10),new Integer(0));    List<Integer> filteredList = filterList(integerList, positive);    filteredList.forEach(System.out::println);   }   public static List<Integer> filterList(List<Integer> listOfIntegers, Predicate<Integer> predicate){    List<Integer> filteredList = new ArrayList<Integer>();    for(Integer integer:listOfIntegers){     if(predicate.test(integer)){      filteredList.add(integer);     }    }    return filteredList;   }  } |

**OUTPUT of the above code**

1  10  200  101

**Explanation of the code and output**

* The static method **filterList()** takes two inputs –
  + A List of Integers which need to be filtered based on some *condition*
  + An instance of Predicate interface which is the condition for evaluation of each integer passed in the Integer list.
* The **filterList()** method loops through the whole IntegerList and whichever integer passes the condition *test* it is added to the resultant list called **filteredList**.
* The caller of **filterList()** gets a *filtered* list back which contains all the Integers which satisfy the *test condition* i.e. greater then zero or *positive*. This is exactly the output we saw above, i.e. **1 10 200 101**
* A lambda expression **i -> i > 0** assigned to an instance of type Predicate named *positive*.
* This predicate instance **positive** is then passed as the second argument of **filterList()** method
* The input **Integer** list can thus be filtered in a different way by writing a *new*lambda with a new test condition(such as less than zero, greater than 100 etc) and the **filterList()** method will apply the test condition to the input list passed. Thus, the test condition is passed as a parameter to the **filterList()**method using the **Predicate** interface

**Default methods in java.util.function.Predicate**: There a few default methods also provided in the Predicate functional interface which enable us to do various types of boolean operations such as and, or, not(negate) with different instances of Predicate. These default methods are –

| **Default Method Name** | **Explanation** |
| --- | --- |
| **and()** | It does logical AND of the predicate on which it is called with another predicate. Example: **predicate1.and(predicate2)** |
| **or()** | It does logical OR of the predicate on which it is called with another predicate. Example: **predicate1.or(predicate2)** |
| **negate()** | It does boolean negation of the predicate on which it is invoked. Example: **predicate1.negate()** |
| Where, **predicate1** & **predicate2** are instances of Predicate interface/lambda expression/method references | |

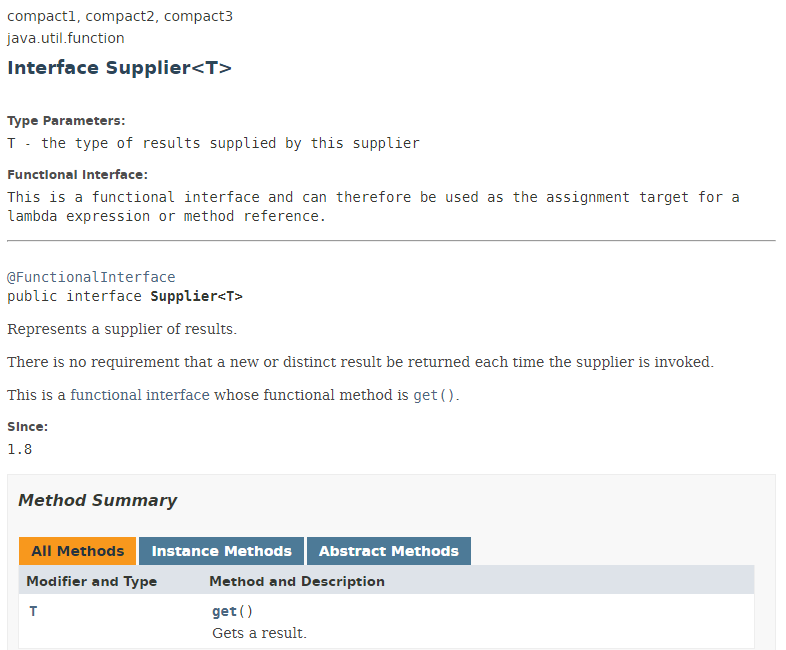
Lets say we have the same code as used above for the filterList() method. I.e. filterList will filter the input Integers List based on the *predicate*passed to it.  
What we are going to change is we will pass different predicates ANDed, ORed, Negated using the default methods and see the resultant output in the table below *(Note – I am assuming that all readers are aware of AND/ OR/ NOT operations in boolean logic)*

| **Predicate passed** | **Values in filteredList** | **Explanation** |
| --- | --- | --- |
| Lets say values in input IntegerList are – [-10, 0, 1, 20, 101, 200] | | |
| **(i->i>0).and(i->i>10)** | [20,101,200] | Conditions for i greater than 0 and i greater than 10 leaves us with 3 filtered values 20,100 & 200. |
| **(i->i>0).or(i->i>10)** | [1,20,101,200] | Either i is greater than 0 or i > 10 gives us 4 filtered values 1,20,101 & 200. |
| **(i->i>0).negate()** | [-10,0] | Negation of i greater than zero implies i is less than or equal to 0. This gives us 2 values -10 & 0 |

**Static method isEqual()** – Predicate interface also has a static method **isEqual()** which can be used to compare 2 instances of Predicate functional interface.  
Its defined as – **static <T> Predicate<T> isEqual(Object targetRef)**  
It returns a predicate that tests if two arguments are equal according to **Objects.equals(Object, Object)**.

**Summary** – We looked at what is java.util.function.Predicate functional interface, what is the advantage of having this interface, we looked at the source including the abstract method, then we looked at the default methods, saw examples of their usages and finally learned about the static method present in the Predicate Interface.

### Supplier



Java 8 java.util.function.Supplier

**Introduction**: Tutorial explains the in-built functional interface **Supplier<T>** introduced in Java 8. It explains with the help of examples how the Supplier interface is to be used via its **get()** method.

**What is java.util.function.Supplier**: **Supplier<T>** is an in-built [functional interface](https://www.javabrahman.com/java-8/functional-interfaces-java-8/) introduced in Java 8 in the **java.util.function** package. Supplier can be used in all contexts where there is no input but an output is expected.

Since Supplier is a functional interface, hence it can be used as the assignment target for a [lambda expression](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/) or a [method reference](https://www.javabrahman.com/java-8/java-8-method-references-tutorial-examples/).

**Function Descriptor of Supplier<T>**: Supplier’s Function Descriptor is **() -> T**. This means that there is no input in the lambda definition and the return output is an object of type T. To understand Function Descriptors in details you can refer the [function descriptor tutorial](https://www.javabrahman.com/java-8/function-descriptors-java-8-explained/).

**Advantage of predefined java.util.function.Supplier**: In all scenarios where there is no input to an operation and it is expected to return an output the in-built functional interface **Supplier<T>** can be used without the need to define a new functional interface every time.

**java.util.function.Supplier source code**

**java.util.function.Supplier source code**

|  |
| --- |
| @FunctionalInterface  public interface Supplier<T> {      /\*\*       \* Gets a result.       \* @return a result       \*/      T get();  } |

**Salient Points regarding Supplier<T>’s source code**:

* **Supplier**has been defined with the generic type **T** which is the same type which its **get()** methods return as output.
* **get()** method is the primary abstract method of the Supplier functional interface. Its function descriptor being **() -> T**. I.e. **get()** method takes no input and returns an output of type T. I will explain usage of **get()**with detailed example in the next section.
* All lambda definitions for Supplier must be written in accordance with **get()** method’s signature, and conversely all lambdas with the same signature as that of **get()** are candidates for assignment to an instance of Supplier interface.

**Usage of get() method of Supplier**:  
To understand the **get()** method lets take a look at the **SupplierFunctionExample**’s code below, post which I have explained in detail how the code works-

**Code showing usage of Supplier.get() method**

|  |
| --- |
| //SupplierFunctionExample.java  import java.util.Date;  import java.util.function.Supplier;  public class SupplierFunctionExample {   public static void main(String args[]) {    //Supplier instance with lambda expression    Supplier<String> helloStrSupplier = () -> new String("Hello");    String helloStr = helloStrSupplier.get();    System.out.println("String in helloStr is->"+helloStr+"<-");      //Supplier instance using method reference to default constructor    Supplier<String> emptyStrSupplier = String::new;    String emptyStr = emptyStrSupplier.get();    System.out.println("String in emptyStr is->"+emptyStr+"<-");      //Supplier instance using method reference to a static method    Supplier<Date> dateSupplier= SupplierFunctionExample::getSystemDate;    Date systemDate = dateSupplier.get();    System.out.println("systemDate->" + systemDate);   }   public static Date getSystemDate() {    return new Date();   }  } |

**OUTPUT of the above code**

String in helloStr is->Hello<-  
String in emptyStr is-><-  
systemDate->Wed Dec 16 19:18:15 IST 2015

**Explanation of above example’s Code & Output**

* **SupplierFunctionExample** is my class with 2 methods – **main()** & **getSystemDate()**.
* **getSystemDate()** is a static method which simply returns the current system date and does not take any input. The method signature matches the function descriptor of Supplier i.e. **() -> T**.
* **In main() method I have shown how to instantiate a Supplier interface instance in following 3 ways**–
  1. **Using a Lambda Expression**: I have defined a a lambda expression which takes no input and returns a new String object with value set to “hello”. This lambda I have assigned to a **Supplier<String>**instance named **helloStrSupplier**. Invoking functional method **get()** on **helloStrSupplier** gives us a String **helloStr** which is then printed to show that it indeed contains the value “hello”.
  2. **Using a Method Reference to default constructor of String**: Method Reference to the default constructor of **String** is used to create a **Supplier<String>** instance named **emptyStrSupplier**. **emptyStrSupplier** is then used to create a String object named **emptyStr** using the **get()** method. **emptyStr**‘s value is then printed to show its value is empty as defined.
  3. **Using a Method Reference to getSystemDate()**: Method Reference to the **getSystemDate()**method of **SupplierFunctionExample** class is used to create a **Supplier<Date>** instance named **dateSupplier**. **dateStrSupplier** is then used to create a **Date** object named **systemDate** by invoking **get()** method on it. **systemDate**‘s value is then printed to show its value. Value printed is of 16-Dec when I ran this example.

**Summary**: In this tutorial we looked at what is the **Supplier<T>** in-built interface defined in Java 8 and what is its main advantage. We then looked at how to use the **Supplier<T>** interface using its **get()** method with an example.